Genium Data Store: Distributed Data Store

IULIIA PROSKURNIA

Master’s Degree Project
Stockholm, Sweden June 2013

TRITA-ICT-EX-2013:182
Genium Data Store: Distributed Data Store

IULIIA PROSKURNIA

Master’s Thesis
Supervisor: Björn Carlson and Jim Dowling
Examiner: Johan Montelius

TRITA-ICT-EX-2013:182
Abstract

In recent years the need for distributed data storage has led the way to design new systems in a large-scale environment. The growth of unbounded stream of data, the necessity to store and analyze it in real time, reliably, scalably and fast are the reasons for appearance of such systems in financial sector, stock exchange Nasdaq OMX especially.

Futhermore, internally designed totally ordered reliable message bus is used in Nasdaq OMX for almost all internal subsystems. Theoretical and practical extensive studies on reliable totally ordered multicast were made in academia and it was proven to serve as a fundamental block in construction of distributed fault-tolerant applications.

In this work, we are leveraging NOMX low-latency reliable totally ordered message bus with a capacity of at least 2 million messages per second to build high performance distributed data store. The data operations consistency can be easily achieved by using the messaging bus as it forwards all messages in reliable total order fashion. Moreover, relying on the reliable totally ordered messaging, active in-memory replication support for fault tolerance and load balancing is integrated. Consequently, the prototype was developed using production environment requirements to demonstrate its feasibility.

Experimental results show a great scalability, and performance serving around 400,000 insert operations per second over 6 data nodes that can be served with 100 microseconds latency. Latency for single record read operations are bound to sub-half millisecond, while data ranges are retrieved with sub-100 Mbps capacity from one node. Moreover, performance improvements under a greater number of data store nodes are shown for both writes and reads. It is concluded that uniform totally ordered sequenced input data can be used in real time for large-scale distributed data storage to maintain strong consistency, fault-tolerance and high performance.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contents</strong></td>
<td>ii</td>
</tr>
<tr>
<td><strong>List of Figures</strong></td>
<td>iv</td>
</tr>
<tr>
<td><strong>List of Acronyms and Abbreviations</strong></td>
<td>vii</td>
</tr>
<tr>
<td><strong>List of Definitions</strong></td>
<td>ix</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Nasdaq OMX</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Contribution</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Limitations</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Results</td>
<td>5</td>
</tr>
<tr>
<td>1.6 Structure of the document</td>
<td>6</td>
</tr>
<tr>
<td><strong>2 Background and Related Work</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 NASDAQ OMX Messaging System</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1 MoldUDP</td>
<td>9</td>
</tr>
<tr>
<td>2.1.2 SoupTCP</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Uniform Reliable Total Order Multicast</td>
<td>11</td>
</tr>
<tr>
<td>2.3 LEDS Data Store</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Related Works</td>
<td>14</td>
</tr>
<tr>
<td><strong>3 Genium Data Store</strong></td>
<td>17</td>
</tr>
<tr>
<td>3.1 NASDAQ OMX Domain</td>
<td>17</td>
</tr>
<tr>
<td>3.2 Requirements</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Usage Scenarios</td>
<td>19</td>
</tr>
<tr>
<td>3.3.1 Time Oriented</td>
<td>19</td>
</tr>
<tr>
<td>3.3.2 Object Oriented</td>
<td>19</td>
</tr>
<tr>
<td>3.4 System Design</td>
<td>20</td>
</tr>
<tr>
<td>3.4.1 Functionality</td>
<td>21</td>
</tr>
<tr>
<td>3.4.2 Data and Query Model</td>
<td>21</td>
</tr>
<tr>
<td>3.4.3 Read Query Support</td>
<td>22</td>
</tr>
</tbody>
</table>
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.4</td>
<td>Towards Consistency, Availability and Speed</td>
<td>23</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Low Latency</td>
<td>24</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Partitioning</td>
<td>25</td>
</tr>
<tr>
<td>3.4.7</td>
<td>Replication</td>
<td>25</td>
</tr>
<tr>
<td>3.4.8</td>
<td>Failure Scenarios</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Summary</td>
<td>30</td>
</tr>
</tbody>
</table>

### 4 Software Design

<table>
<thead>
<tr>
<th>4.1</th>
<th>Architecture</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.1</td>
<td>Logical View</td>
<td>31</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Process View</td>
<td>32</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Physical View</td>
<td>33</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Activity Diagrams</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Implementation Details</td>
<td>37</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Client API</td>
<td>37</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Bootstrapping the system</td>
<td>39</td>
</tr>
</tbody>
</table>

### 5 Evaluation

<table>
<thead>
<tr>
<th>5.1</th>
<th>Prototype</th>
<th>41</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Performance</td>
<td>42</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Inserts</td>
<td>43</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Get - Single Record Read</td>
<td>46</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Range Query</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>Fault Resilience</td>
<td>50</td>
</tr>
</tbody>
</table>

### 6 Discussion

<table>
<thead>
<tr>
<th>6.1</th>
<th>Main findings</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.1</td>
<td>Performance</td>
<td>51</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Consistency, availability, failure resilience</td>
<td>52</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Summary</td>
<td>53</td>
</tr>
<tr>
<td>6.2</td>
<td>Positioning</td>
<td>53</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Comparison to existing systems</td>
<td>53</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Stream Processing Applicability</td>
<td>54</td>
</tr>
<tr>
<td>6.3</td>
<td>Future Work</td>
<td>55</td>
</tr>
<tr>
<td>6.3.1</td>
<td>System Improvements</td>
<td>55</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Research Directions</td>
<td>56</td>
</tr>
</tbody>
</table>

### 7 Conclusions

### Bibliography
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Simplified Genium INET multicast</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Total Order Multicast</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Leds data store in-memory organization</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Leds Index Map</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Peak Message Load per Second over Time in NOMX</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>GDS High Level Infrastructure</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Two types of partitioning</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Active replication based on the atomic broadcast primitive</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Failure scenarios during the communication in general</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Possible client failure and its resolution</td>
<td>28</td>
</tr>
<tr>
<td>3.7</td>
<td>Possible sequencer failure and its resolution</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Logical view of the GDS architecture</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Process View of the GDS architecture</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Physical View of the GDS architecture</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>User application, sequencer and data store activity diagrams</td>
<td>35</td>
</tr>
<tr>
<td>4.5</td>
<td>GDS Client activity diagram</td>
<td>36</td>
</tr>
<tr>
<td>4.6</td>
<td>Append and insert</td>
<td>38</td>
</tr>
<tr>
<td>4.7</td>
<td>Update</td>
<td>39</td>
</tr>
<tr>
<td>4.8</td>
<td>Get and range query</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>Prototype Set Up</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Insert operations throughput for 100 bytes messages increasing number of concurrent users</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>Data nodes scalability for insert operation with twice and trice number of requests per node</td>
<td>45</td>
</tr>
<tr>
<td>5.4</td>
<td>Insert operations minimum, maximum and average latency depending on the number of concurrent users in the system</td>
<td>45</td>
</tr>
<tr>
<td>5.5</td>
<td>Get operation latency over filled data store</td>
<td>46</td>
</tr>
<tr>
<td>5.6</td>
<td>Get operation latency with an additional load of 10 000 inserts per second</td>
<td>47</td>
</tr>
<tr>
<td>5.7</td>
<td>Scalability of a get operation in terms of latency reduction over multiple data nodes</td>
<td>48</td>
</tr>
<tr>
<td>5.8</td>
<td>Range query throughput and scalability</td>
<td>48</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>5.9</td>
<td>Range query scalability under constant load of 8 concurrent users for different replication factor</td>
<td>49</td>
</tr>
<tr>
<td>5.10</td>
<td>Range query TCP connection failure with re-request of all or only missed records</td>
<td>50</td>
</tr>
<tr>
<td>6.1</td>
<td>On-line stream processing within GDS infrastructure</td>
<td>55</td>
</tr>
</tbody>
</table>
List of Acronyms and Abbreviations

UC  Use Case
NOMX  NASDAQ OMX
ACID  Atomicity, Consistency, Isolation, Durability
DS  Data Store
DSS  Distributed Data Storage
INET  Genium INET Messaging Middleware
GDS  Genium Data Store
SLA  Service Level Agreement
TO  Total-order |item|CAP| Consistency, Availability, Partitioning
OMX  Swedish-Finnish company of financial services (Optionsmäklarna/Helsinki Stock Exchange)
List of Definitions

**ACID**  Atomicity, Consistency, Isolation, Durability

**atomicity**  In an atomic transaction, a series of database operations either all occur, or nothing occurs.

**consistency**  A consistent transaction is one that does not violate any integrity constraints during its execution.

**isolation**  Isolation is a property that defines how/when the changes made by one operation become visible to other concurrent operations.

**durability**  Durability is the property which guarantees that transactions that have committed will survive permanently.

**atomic commitment protocol**  An atomic commit is an operation in which a set of distinct changes is applied as a single operation.
Chapter 1

Introduction

At Nasdaq OMX (NOMX), continuous improvement of the core technologies is the main focus. Main concerns are latency reduction, capacity and reliability improvement. New generation technologies of NOMX are handling over millions messages per seconds at an average speed of sub-100 microseconds, which is the fastest over any exchange or trading system in the world.

Such competitive field requires completely new approach for building distributed data store which is optimized to serve trading environment. Current data store systems impose different approaches to achieve a great performance, however, due to the existence of CAP theorem such systems compromises some of the important properties of the system, e.g., availability, consistency, fault-tolerance. Aforementioned characteristics should not be simply weakened. Therefore, we decide to build a distributed data store based on NOMX INET messaging technology, a uniform reliable totally ordered multicast middleware. Leveraging this system provides harmonic combination of the desired properties in a store.

We suggest an architecture of a distributed data storage designed to serve millions of user messages, in great capacity data store using reliable NOMX messaging technologies.

1.1 Nasdaq OMX

NASDAQ OMX (NOMX) is the world’s largest stock exchange technology provider. Around ten percent of all worlds’ securities transactions are happening here. More importantly is that, NOMX multi-assets trading and clearing system is the world’s fastest and scalable one. This enables the trading of any asset in any place in the world. NOMX owns 70 stock exchanges of 50 countries and serves multiple assets including equities, derivatives, debts, commodities, structured products and ETFs.

The core messaging system of the company for trading and clearing is called Genium INET and is very complex, optimized and customized for different customers.

A short summary of the company is the following:

- NOMX unites 3900 companies in 39 countries with $4,5 trillion in total market
value.

- NOMX is the world’s largest IT industry exchange.
- NOMX is the world’s largest biotech industry exchange.
- NOMX is the world’s largest provider of technology for the exchange industry.
- Over 700 companies use more than 1500 different corporate products and services.
- NOMX data accessed by more than two millions users in 83 countries.
- The average transaction latency on NOMX in 2010 was less than 100 microseconds.
- The NOMX implementation of INET is able to handle more than one million transactions per second.

In other words, NOMX globally covers the widest geography with its large number of company listings and executes more transactions than any other exchange.

### 1.2 Motivation

There are many existing distributed systems, which are focused on optimization of various systems properties, e.g., availability, robustness, consistency. Designing of a distributed data storage for real time stock exchange environment is very challenging as it should meet strict performance requirements.

First, the application must be responsive; hence, the system should perform with low latency. Second, the robustness criteria are quite challenging as any millisecond of failure is equivalent to huge losses for the customers. Third, the trading environment brings a huge variety of potential users, so the application must be highly scalable. Fourth, the application should provide the user with a consistent view of the data all the time – the result of any operation should be visible immediately and durably. Fifth, it should be up 24/7, so the application must be highly available. Finally, a wide range of operations should be supported, such as insertions, update, single record read (get), and range reads.

Current general purpose solutions are eager to sacrifice some properties (e.g., consistency) in order to achieve great improvements in the other ones (e.g., performance, availability). Moreover, none of them leverages a uniform reliable total order multicast properties [2] to supply fault-tolerant, fast and ACID properties for the operations over data. Increased interest in NoSQL data storage has boosted the appearance of various state-of-the-art systems. Representative work includes BigTable, Dynamo, Cassandra, Yahoo PNUTS [20, 24, 37, 21] etc. Some of them tries to reach performance implying some reduction on consistency or availability requirements, e.g., eventual consistency [37], availability reductions [19, 36]. Some
of them reduce performance using complicated conflict or failure resolution or up-
date propagation. Some are focused on security [11], geo-replication [22], ACID
properties [51, 26] or latency preservation [45]. However, in this work, real time low
latency system performance should meet ACID properties on any operation in the
system. Consequently, previous approaches for the distributed data store (DDS)
arquitectures should be questioned.

Despite algorithmic advancements in total order broadcast and the develop-
ments of distributed database replication techniques, limited research on applying
these algorithms for large-scale real-time data storage exists. One of the most com-
mon usage of the total order broadcast appeared in database replication [54]. This
technique guarantee message delivery reliably and in the same order on all replicas.
Moreover, it requires only O(1) network interaction and do not rely on an atomic
commitment protocol.

Limited application of the total order techniques in the real-time large-scale
systems might be due to the scalability issues and its limited capacity.

We are proposing a distribute in-memory data store system, based on the NAS-
DAQ OMX low latency uniform reliable totally ordered message middleware, which
provides scalability, as the capacity of the message bus exceeds 2 million messages
per second. This messaging abstraction interprets unordered incoming stream of
data into an ordered sequence of operation which are backed up by rewinders and
therefore, message gap-filling mechanism automatically supported and served by
them. An ordered stream of data is published on the, so called, “message stream”
and is seen by everyone on the stream. Based on this messaging infrastructure,
optimistic delivery can be assumed. In other words, an early operation indication is
preserved and it is guaranteed to commit eventually all messages in the same order
to all subscribed servers.

The main focus of this work is to leverage the uniform reliable total order multi-
cast protocol for building real time, low latency, fault-tolerant and ACID distributed
data store system. We have no intension to support complicated joins or multiple-
operation transactions in this project. The major difficulty is to be able to guarantee
fault-tolerance and ACID properties for the operations. Even though maintaining
the system in real time is challenging, supporting range reads and concurrent up-
dates are no straightforward endeavor as well. To reach the performance goals, the
following approach is applied:

- Scalability: Adding extra instances to the system is very easy. The only
  thing that is required is to declare schemas that are served by the data store.
  Schema here is a declared set of columns within some partition in the data
  store.

- Availability: Ability to serve request at any given time is provided for all op-
erations, e.g. reads, writes. First, the capacity of the messaging middleware is
  claimed to be enormous, due to the number of tweaks made to the technology.
  Second, range requests responses are sent directly to the requester and are
served by the fastest replica, therefore, the load pressure is reduced to the sequencer and the system.

- Consistency: As the underlying message passing abstraction produces a uniform reliable totally ordered stream of requests, each instance observes exactly the same sequence of messages. This gives a consistent view to any instance at any requested time. Similarly for concurrent updates, totally ordered timestamps per update are used, hence timestamp concurrency control [15] is deployed.

- Fault-Resilience: As absolutely equal stream of requests are received by any of the replica, failure of any instance during operations is transparent. Failure of the data store or connection during the range query serving is handled by the snapshot indication message on the message stream. This way the query can be requested again from the fracture place and replayed till the snapshot point.

- Query Support: In order to increase the availability level, limitation on the query response is set. If the response exceeds the limit of 100 000 records per request, the query should be submitted again for the desired records.

1.3 Contribution

There exist limited research on the large-scale real-time distributed data store system based on uniform reliable total order multicast. At the same time, the NOMX INET messaging middleware shows a very promising performance characteristics. Considering mentioned above, the system that serves hundreds of thousand requests per seconds with low latency and in reliable and consistent fashion was designed. The system elicit a way to combine low latency, scalability, consistency and fault-tolerance with various operations support, e.g., insert, update, single record read (get), range read. Taking into account strict stock exchange desire not to dedicate its budget to the expensive commercial solutions that still lack the real-time properties, our proposed system have a great chance to hold a niche in this company. Moreover, our system provides a user with near sub-100 microseconds latencies for inserts and sub-400 microseconds for reads over millions of records.

The system is agnostic to the consistency problems as all messages for all the nodes appear reliably in the same order.

The primary use-case for the proposed distributed data store are confined of 1) logging the information in a time oriented fashion with a possibility to read time ranges and 2) storing objects with possible further updates and get. Hence, the majority of examples throughout the report considers these two scenarios. More detailed information can be found in the Section 3.3.

System prototype handles on average over 250 000 different requests per second while scaling almost linearly by adding additional data nodes and able to handle various failures.
Consequently, this report contains 4 primary contribution. First, an overview of existing systems, which are used for large-scale storage usage based on different approaches, e.g., total order multicast, active replication, concurrency control, eventual consistency models etc. Second, a conceptual approach and demonstration of how uniform reliable total order multicast can be used to create highly scalable consistent, fast and reliable distributed data storage system at large scale to serve real time requests. Third, the simplicity of the stream processing based on the developed system is discussed. Finally, the prototype implementation which proves the satisfaction of the performance requirements was built.

To our knowledge, GDS is the first low latency, asynchronous, scalable, strongly consistent, fault-tolerant and fully distributed key-value store without a shared consensus component and high extensibility to on-line stream processing support.

1.4 Limitations

Prototype’s features were prioritize and filtered by the most to show GDS prototype performance. Furthermore, simple column based database notation is only supported without going into complex joins and group by on the data. Instead, the data is partitioned, as it is discussed in the section 3.4.6 according to its nature and optimality for fast read operation on it, that are specified in section 3.3.

For simplicity, transactions over data are reduced to the single operation per transaction. GDS is not intended to serve complex multi operational transactions and join operation.

Moreover, reliance on the reliable total order multicast assumes that the multicast is provided in the uniform fashion. Therefore, in case of hardly possible violation of this property, no functionality to rollback the transaction is handled by the prototype, but is a question of a future work.

Lightweight and flexibility of the system architecture can easily handle disperse variety of supported data stores, e.g., key value, column oriented, SQL-like, however low priority of this option influenced the choice of only one supported column oriented data store. This data store is developed in place and optimized for fast inserts. More complete and detailed information about the used database is presented in the Section 2.3.

1.5 Results

To show the proof of concept for the GDS system a set of experiments on performance was made. Insert, get, query operations performance was measured and presented through throughput and latency indication. Moreover, a scalability of the system was tested by adding further data nodes. The main purpose of these test is to show a great performance over non dedicated hardware and experiment on the system scalability. Finally, a simple test for the query transmission failure
resolution was made and it showed 15% gain in speed in comparison to the case were failed requests were just rerequested.

Performance tests show a great throughput for the write operations of 400 000 inserts per seconds over 6 machines with sub-100 microseconds latency. Moreover, all inserts are actively replicated, consistent at any point in time and robust to failures. This properties makes the latency and throughput indicators unique. Get operation performance has shown sub-500 micro seconds lookup latencies over several million records data base.

The main factors on the way combine both consistency and speed was the proper leverage of the NOMX messaging middleware that guarantees uniform reliable total order delivery of all messages. As the sequencer is the "heart" of the GDS, it was designed to only order incoming stream of messages into an totally ordered sequence of requests.

Overall, the system has shown a great performance in terms of throughput and latency. However, a long running tests over greater amount of nodes should be made to explore how the performance is affected over time.

1.6 Structure of the document

The report starts with the background to GDS (Section 2), primarily those of interest as building blocks for the system. Following an overview of the related work (Section 2.4) with explicit highlighting of the main distributed system properties and systems’ positioning among those properties. The subsequent section (Section 3) explains the most important design concepts that were chosen for the system. Proposed architecture and API are presented in the Chapter 4. Experimental performance, scalability and failure resilience results are presented in the Chapter 5, with a follow up discussion in the Section 6. Finally, section 7 summarizes the main finding s and concludes the results.
Chapter 2

Background and Related Work

GDS is a complex and compound system. In this chapter the most relevant composition parts of the system is introduced, e.g., messaging middleware, pluggable back-end storage. Moreover related work is presented in the end of the chapter.

2.1 NASDAQ OMX Messaging System

To achieve performance goals, GDS leverages high-performance of GENIUM INET messaging bus. GENIUM INET messaging bus is based on UDP multicasting and is made reliable by totally ordered sequence of backed up messages with a gap-fill mechanism using rewinders. As discussed, a total order broadcast algorithms are fundamental building blocks for construction of fault-tolerant systems, as described in the next section 2.2. The purpose of such algorithms is to provide communication primitives that allow processes to agree on the set of messages and order that deliver. NASDAQ OMX implementation of this abstraction assumes a perfect failure detector, e.g. it forces a process to fail if it was considered faulty. Moreover, uniform reliable total order is preserved, where a process is not allowed to deliver any message out of order, even if it is faulty. NOMX techniques used to achieve uniformity and its evaluation is out of the scope of this project.

Messaging middleware has a single member that assigns sequenced message identifiers. This member is called sequencer. Figure 2.1 illustrates the role of a sequencer in the transmission of a multicast message. The originator of a multicast message issues a request that is sent to the sequencer first. It assigns a number and replays it to the other members using a single multicast message. Multicast is presented in an atomic manner, that is, when a crashed replica recovers, it is forced to bring its state up to date with the rest of the multicast group. Consequently, all non faulty processes maintain a consistent view of the system and require reconciliation upon recovery. Even though the communication between processes is asynchronous the reliability is provided by the message gap filling mechanism upon message loss.

Among receivers of the message are general nodes, standby sequencer and rewinders. Rewinders keep track of the messages replayed by the sequencer and are able to
Figure 2.1: Simplified Genium INET multicast

Rewind the message stream on request. This component drastically reduce the amount of information stored in the sequencer, so that, the sequencer does not need to store all message history. Moreover, data on rewinders decreases load to the sequencer in case of failures and need to fill the message gaps on the client side. However, there is one constraint on the rewinder configuration, that is, for the message bus to work well the system configuration must be calibrated to not overload rewinders. It is recommended to configure the number of rewinders according to the possible number of concurrent nodes that requires rewinding. Otherwise unrecoverable packet loss may occur and hence violate the transaction semantics.

The receivers of the ordered messages should guarantee exactly-once delivery to the applications for each message, consequently uniform integrity is guaranteed. Across the cluster of applications/clients connected to the message stream, messages should be delivered in the same order.

The message stream can be configured without restriction, so that it can contain any number of various clients that can be placed on the same or different server, with defined replication level. The sequencer is responsible for the sequenced/ordered stream creation as an output from received clients’ messages.

Failure resilience is provided by the following:

- All message callbacks are fully deterministic and re-playable (if single-threaded),
as the incoming stream is identical each time it is received.

- Replication can be adopted by installing the same receivers at multiple servers.
- As long as the new primary rewound to the same point as the failed one, the message stream is sufficient to synchronize state.

GDS uses the privileges of the described above infrastructure and a priory maintains a fault-tolerant real-time system. Moreover, the distributed state is made consistent by adhering to the sequenced numbering implied by the message stream.

**Functionality and API**

The main function of the following messaging middleware is ensure reliable and total order transmission of messages from a sender to all subscribed members in the systems. This functionality dictates the API.

The client API for the messaging system is simple and straightforward. It allows the applications to connect to the stream and co-exist with all applications on the stream, both native and Java. The API is based on using handles, as a source of the information and callbacks, as a communication points. Main functions supported:

- **Client initialization** provides client registration as a subscriber to the sequencer message stream.
- **Client send** supply a client with the opportunity to send messages to sequencer.
- **Client on event** offer a point of communication from the sequencer to the client direction.
- **Server initialization** provides a server declaration of being a sequencer.
- **Server send** offer a functionality to multicast outcoming ordered messages to the nodes subscribed to the sequencer.
- **Server on event** supply an event driven point of communication to sequencer from the client.

Note that here, clients are the application placed on the stream and a server is a sequencer.

### 2.1.1 MoldUDP

In a GENIUM Data Store, transactions are submitted through the MoldUDP, therefore it ensures the lowest possible transaction latency. MoldUDP is a networking protocol that makes transmission of data messages efficient and scalable in a scenario where one transmitter and many listeners are present. It is a lightweight protocol that is built on top of UDP where missed packets can be easily traced and detected, but retransmission is not supported. Some optimization can be applied
to make this protocol more efficient: (a) multiple messages are aggregated into a single packet - to reduce network traffic, (b) caching Re-request Server is placed near remote receiver - to reduce the latency and bandwidth.

MoldUDP presumes that system consists of listeners, which are subscribed to some multicast groups, and server, which transmits on those multicast groups. MoldUDP server transmits downstream packets through UDP multicast to send normal data stream addressed to listeners. MoldUDP server sends heartbeats periodically to clients, so they can retrieve information about packet loss if it takes place. Moreover, listeners should be configured with IP and port to which they can submit the requests.

Note: message in this context is an atomic piece of information that is carried by the MoldUDP protocol from 0 to 64 KB.

2.1.2 SoupTCP

In GENIUM Data Store, read query support will be maintained. That is why, TCP-like protocol is intended to be used to stream the data to the client in response to the submitted query.

SoupTCP is a lightweight point-to-point protocol build on top of TCP/IP sockets. This protocol allows delivering a set of sequenced messages from a server to a client. It guarantees that the client will receive all messages sent from a server strictly in sequence even when failures occur.

Server functionality with SoupTCP includes: (a) clients authentication on login and (b) delivery of a logical stream of messages to a client in a real-time scenario. A client sends messages to a server that are not guaranteed to be delivered in case of failures. That’s why the client will need to resubmit the request to the server.

Protocol flow:

- Client opens a TCP/IP socket to the server with login request.

- If the login information is valid - server responds with accept and starts to send sequenced data.

- Both client and server compute message number locally by simple counting of messages and the first message in a session is always 1.

- Link failure detected by the hear beating. Both server and client send these messages. Former is required to notify a client in case of failure to reconnect to another socket. Later is necessary to close the existing socket with failed client and listen for a new connection.
2.2 Uniform Reliable Total Order Multicast

Multicast operations generate messages that are sent from one process to a set of processes and the membership of a group is usually transparent to a sender [23]. However simple multicast protocol does not guarantee any ordering or message delivery. Therefore, stronger assumptions should be made in a frame of nowadays distributed systems, such as, reliability. GDS relies on the reliable multicast, in which any transmitted message is either received by all or none of the processes. In other words, there could not be a situation where a client accesses a server just before it crashes and observes an update that no other server will process. This property is called uniform agreement. Moreover, to maintain a consistent and fault-tolerant system a total order assumption should be made additionally to reliable uniform multicast.

The simplest specification of the uniform reliable total order multicast can be defined in terms of two primitives [25], which are TO-multicast(m) and TO-deliver(m), where m represent a message. When a process issued a uniquely identified message m as TO-multicast(m), this assumes following properties [30]:

- **Validity.** If a correct process TO-multicast a message m, then it eventually TO-delivers m.
- **Uniform Agreement.** If a process TO-delivers a message m, then all correct processes eventually TO-deliver m.
- **Uniform Integrity.** For any message m, every process TO-delivers m at most once, and only if m was previously TO-broadcast by the sender.
- **Uniform Total Order.** If two processes, p and q, both TO-deliver message m and m’, then p TO-deliver m before m’, if and only if q TO-delivers m before m’.

If all these properties are satisfied then reliable total order multicast takes place. NOMX messaging middleware cover all this properties and therefore, GDS is build on top of the uniform reliable total order multicast. Uniformity in the system is presented as not allowance to deliver a message out of order by any process. This is quite strong statement which goes beyond this thesis project and provided and proved in production by the NOMX.

Internally, multicast communication is used for most of the subsystems as it is the only fast and reliable way to guarantee consistency and agreement within all nodes with minimal cost.

Although there are two main ways to maintain total order, i.e., consensus, sequencer based [33]. GDS uses the single sequencer ordering mechanism as the more efficient in comparison to the consensus one. The simplest presentation of the total order ordering is illustrated on the Figure 2.2. This figure shows that no matter when the messages were issued they will be delivered in the same order to all the processes. For the sequenced mechanisms the main problem is a possible bottleneck
and critical point of failure in sequencer part. It can be overcomes using the replicated standby sequencer that is delivers all messages issued by the primary one and takes over in case of failure.

![Figure 2.2: Total Order Multicast](image)

## 2.3 LEDS Data Store

LEDS is a data store that is developed inhouse by NOMX. The main purpose of LEDS data store is to manage very large amount of structured data. It may contain billions of records and optimized for very fast inserts and searches. All data in the store is time-indexed and therefore, searches are made faster with time occurrence of the specific data. The data store can handle around 100 000 operations per second on billion record data store.

The data store is build in Java and the data stored as arrays of compressed chunks. Compression might be optionally applied only when saving data to disk. The prototype version used ZLIB compression which guarantee the integrity for the in memory compression and decompression. Each chunk is an array of records and keys, as it is shown on the Figure 2.3. LEDS does not support a full relational data model; instead, it provides a simple data model with defined number of columns and defined column sizes. This simplicity allows to search the data efficiently, simple scanning the key index within columns. The index maps’ keys to any chunk where there is at least one record with the same key. As each chunk has a limited size, index mapping reduces scanning space to several order of magnitude. That is, using the index mapping search is reduced to the particular chunk of limited size. Additionally to the key-chunk index, time-index is applied to the records. All of this provide better searches and better compression, while being stored in the memory.

Basic API provides the following operations:

- Create schema
- Add records
Search the record from keys
Manage to disk

Key conjunction is searched by mapping each key separately to a list and then intersecting lists. Key disjunction should be implemented as two different searches.

Index map is a trie-map from a key of bytes to a list of chunks which is one per column and data store. A tree is build from arrays, where each element points to null or a subtree or a leaf, as shown on the Figure 2.4.

Overall, LEDS provide a solid foundation for a real time data mining and can handle large amount of data. The API is quite simple and optimized for the most important operation on the stoke exchange, as append and time range searches.
2.4 Related Works

Recently there has been increasing interest in NoSQL data storage to meet the highly intense demand of the applications. Representative work includes Bigtable, Megastore [14], Cassandra [37] and Yahoo PNUTS [21] etc. In these systems, scalability is achieved by sacrificing some properties, e.g. transactions support, fault resilience or performance. On the other side, most prevailing data storage systems use asynchronous replication schemes with a weaker consistency model, e.g., Cassandra [37], HBase, CouchDB and Dynamo [24] use an eventual consistency model. Conventional database systems provide mature and sophisticated data management features, but have difficulties with serving large-scale interactive applications. Open source database systems such as MySQL do not scale up to required levels, while expensive commercial database systems like Oracle significantly increase the total cost of ownership in large deployments. Moreover, neither of them offers fault-tolerant, e.g., non-expensive replication mechanism which is the key piece to build robust applications.

According to two main usage scenarios, described in Section 3.3, the main difference between other known systems and GDS should be sliced into two parts, respectively. On the one hand, GDS system is optimizing the tuple \{appends, time range queries\} operations simultaneously. On the other hand, it tuned to use tuple \{inserts, updates and record reads\} operations with the best performance.

Since a few years, scalability stops being a unique property of distributed system and should be guaranteed with a high priority [3, 9, 11]. GDS is not an exception and, supposedly, can scale up to hundreds of servers and users concurrently in the system. This guaranteed by the high capacity of the system’s lightweight sequencer and data partitioning over the nodes. To maintain scalability different partitioning approaches are applied, mainly dependently on the further system usage and performance requirements, Section 3.4.6. In the key-value store niche, consistent hashing [34] of the data among the servers is popular [21, 8]. However, it introduces a complexity for retrieving ranges of records. The other approach is to apply range or list partitioning [3, 11], like in document oriented stores, which are responsible to
store the data localized towards time or particular feature, respectively.

One of the conceptual differences from the aforementioned systems is that in the GDS no routing is required to insert or retrieve data, as everyone in the system receives the same data and filters it in place. On the contrary, the most popular systems are lack this approach and therefore, increase the latency. Most of the well known systems are keen on using structured overlays, Chord \cite{49}, Pastry \cite{7}, like in Cassandra \cite{37}, Dynamo \cite{24}, Megastore \cite{14}, Pastry \cite{19}, Riak \cite{4} etc. This introduces messaging overhead in the system in a form of $O(\log N)$ routing, where $N$ is the number of nodes in the system. There are some systems with $O(1)$ routing complexity, however they implies master/slave paradigm with a single point of failure, e.g., Keyspace \cite{51}. At the same time, the data search can be optimized using traversal trees stored on the master node \cite{18}.

Consistent replication has been provided by Dynamo \cite{24}, Megastore \cite{14} and Spanner \cite{22}. Dynamo offers a region replication, where Megastore and Spanner tries to cover geo-replication. GDS also offers consistent replication over the data store but there is no support for the geo replication of the data, even though the whole system can be replicated to handle an unexpected disaster. Spanner is the system optimized for geo-placement and it uses atomic and GPS clocks to manage consistency in replicas. Another approach is to use a chain replication \cite{52} used in the Warp \cite{26} key value store. On the contrary, GDS uses total orderer to supply replication with a single synchronization "barrier", e.g., sequencer.

Another important issue is fault-tolerance and it is highly prioritize and critical part of the distributed computing and distributed datastore design in particular. One solution for achieving fault-tolerance is to rely on a fault-tolerant hardware. However, this solution is considered to be expensive and advised to avoid. Software replication \cite{28} is a great solution to increase system failure resilience and it is less expensive that hardware replication. Some of the solutions to the fault-tolerance problem include increasing use of active \cite{46} and passive \cite{53} replication etc. \cite{34}, overhauling the system so that availability is improved and process crash is handled, applying consensus-like protocols.

However, GDS goes a bit beyond the general approach of the failure resistance. Besides, handling a process and link failure by means of reliable total order multicast \cite{30}, it also uses optimization for range queries performance under failures by two mechanisms: consistent snapshot \cite{3.4} and re-request of the failed query from the point of failure.

Consistency is an important issue in distributed systems. To guarantee consistency, it is a common solution to use general consensus protocols, such as Paxos \cite{38} and Zab \cite{32}. It was widely used in the following systems, Megastore \cite{14}, KeySpace \cite{51}, Scalaris \cite{47}, Spanner \cite{22}, Scatter \cite{27}, Spinnaker \cite{43} etc. Some systems imply quorum like mechanisms for reads and writes \cite{24}, others imply vector clocks \cite{1}. However, for some systems strong consistency guarantees are not prerequisite and therefore, eventual consistency models are preferred, e.g., Cassandra, HBase, CouchDB etc. Consensus approach to reach an agreement among a group of replicas is maintained by two rounds of messages until consensus is reached. This introduces un-
necessary overhead and degrade performance significantly. Another notable systems uses synchronized clocks \cite{22, 12}. GDS has adopted a total order multicast protocol for the consistency guarantees and has avoided complication of the system. Moreover, consistency during concurrent updates is handled using timestamp concurrency control, similar to the Dynamo with its versioning.

Another system that is similar to GDS backbone is a Zab \cite{32}, which uses reliable broadcast for primary backup replication preserving consistency and fault-tolerance; however, number of required phases to reach an agreement and, consequently, performance characteristics of this approach is not suitable for the stock exchange environments.

Availability and scalability is an important aspiration. Partitioning is applied to increase scalability and availability of the system \cite{24, 37, 21} etc. GDS leverages two types of partitioning to optimize both use cases within their space, e.g., range and hash partitioning in time and object use case respectively.

Among the functionality of some well known DS, only limited set of operations is used, such as, put and get, \cite{24, 47} etc. Another data store tries to go further and cover updates and range queries. In \cite{41} evaluation of range queries in key-value stores was made. The problem of range querying is extensively explored for publish/subscribe \cite{29} or distributed hash tables \cite{42}. GDS support two different sets of data with inserts, scan and insert, update, read support respectively. Consequently, the operational set is expanded to cover more operations.

Encountering different properties and their combination, performance and low latency is not always a endeavour. A requirement to serve request within microseconds is not always a compulsory. The main reason for it, is that a priory support for low latency should be enclosed to the architecture and all design decision from the start. This may limit other systems properties which are prioritized higher. In a chase for latency, MySQL Cluster is the one that can meet NOMX performance requirements, however (a) it guarantee eventual consistency among replicas, (b) it would require a change in current NOMX infrastructure. Among techniques to reduce the latency are the following, in memory storage, such as VoltDB, MemcacheDB etc, ssd \cite{10} storage, hardware tweaks.

An example of the data store that oriented towards stock exchange environment is the StockDB \cite{5}. This data store is optimized for inserts and range queries. However, the main focus in StockDB is laid over the specific data types compression and its further forward to the client in compressed fashion. Unlike GDS, where any size ranges can be retrieved from the store. Moreover, fault-tolerance is not a concern in the StockDB.

Combining the merit from both scalable data stores and databases, Genium Data Store (GDS) provides ACID guarantees with high scalability, fault-tolerance, consistency and replication. However in case of GDS, as it was mentioned above, wide-area network semantic is not taken into account, as the range of applications, which are served by GDS, do not require wide-area replication and it introduces extra complexity to the system.
Chapter 3

Genium Data Store

3.1 NASDAQ OMX Domain

Information technology part of NOMX, as a set of systems, processes that interact with each other, has some issues to maintain, such as, high level reliability, low latency and security. This niche is unique and normally not covered among conventional IT vendors and consultancies. The company covers the following interchanging systems: trading systems, clearing systems, market data systems, surveillance systems, advisory systems.

The data from NOMX systems is required to be stored reliably and with an extremely fast response. Users (systems) of the GDS system dictate a design decisions that make a system reliable, available, consistent and fast. Moreover, reads and range query requests over the data are an important feature of a system that serves NOMX needs. Additionally, "on the fly" incoming data processing should be kept in mind while designing a system, so that, the storage could be easily enhanced with 'on the fly' functionality and further serve as a backbone for the future on-line alteration. To operate over both inserts, reads, range queries and on-line data processing, a new approach for designing distributed system must be applied.

![Figure 3.1: Peak Message Load per Second over Time in NOMX](image-url)
Before one defines a system it is important to understand exactly what the purpose of the storage system is. Figure 3.1 shows the enormous system load growth over time. The number of peak requests per second during the last 7 years grew exponentially and reached over one million requests per second. Taking this into consideration the next subsection presents SLA requirements for the GDS.

### 3.2 Requirements

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDS Purpose</td>
<td>To aggregate NOMX data from its various systems reliably, consistently, scalable and fast. Especially a certain pressure is put on the following operations: insert, read, range query, additionally to modify operation support over object oriented data, described in Section 3.3.2.</td>
</tr>
<tr>
<td>Integration with other systems</td>
<td>GDS embedded over NOMX messaging middleware and leverages its properties. External users connect to the data store using GDS API.</td>
</tr>
<tr>
<td>Data Store type</td>
<td>Column oriented data store; consistent, available, reliable and fast.</td>
</tr>
<tr>
<td>Number of concurrent users</td>
<td>&lt; Hundred</td>
</tr>
<tr>
<td>Application infrastructure</td>
<td>An application connects to the GDS through its API, which make GDS architecture transparent to the application.</td>
</tr>
</tbody>
</table>

Table 3.1: GDS description required to fulfill

The system should provide a functionality to insert, update, get and range query the data. GDS is an application that is intended to be a component in the NOMX ecosystem and it is specified in more details in the Table 3.1. Some performance expectations are the following (SLA):

- The GDS system should handle around 250 000 requests per second, where each message is less than 200 bytes.
- A query for a record, with a unique index, should be returned in no more than 10 milliseconds with a 10 billion records database.
- The capacity of a given response stream should be 100 Mbps.
- The response streaming capacity of a datastore server should be 300 Mbps (divided across several clients), while receiving 100 000 messages per second.
- An insert should be acknowledged on the message bus within 100 microseconds.
3.3 Usage Scenarios

3.3.1 Time Oriented

Time oriented schema is intended to serve log records storing from specific services. Keys for columns of defined data schema and the data should be included to the requests. At the same time, records retrieval for a given time interval is supported.

The following schema is considered to be used by the application that collect a lot of data constantly and that require to store this data fast and reliably. The first use cases is a logging of the mobile device motion over time with the following information: user name, the location, the time. Here, each user is a source of data, which are appended steadily to user specific table. Partitioning for this case is described in the Section 3.4.6. Such partitioning is required for the purpose of data lookup and retrieval acceleration.

Another logging application is related to the NOMX trading environment and should serve as a fast storing and range retrieval application for all orders and trades that happens on the market for many years. Consistency, availability and speed are the most important criteria here.

An example of this use case can be represented as following pseudo code:

```
Listing 3.1: Time Oriented Scenario. Pseudocode

1  while (true)
2    Append(table, key, columnName1, .., timestamp)
3    if (iteration%N == 0) // where N is configurable
4      Query(table, timestamp1, timestamp 2) => return records
```

3.3.2 Object Oriented

Object oriented schema should support data inserts and updates, together with gets. For example, it should submit inserts and updates for a trade object with the following information: the identity (trade number), the member, the price, the quantity. Automatically a version/timestamp of the record is assigned. Get should return a record data together with an object version.

```
Listing 3.2: Object Oriented Scenario. Pseudocode

1  while (true)
2    Insert(table, key, column1, .., )
3      if (iteration%K== 0) //where K and M are configurable
4        Get(table, key, columnName) => return (data, version)
5    if (iteration%M == 0)
6      Update(table, key, columnName1, .. timestamp)
```
3.4 System Design

The purpose of the following section is to reveal the system organization and properties.

Figure 3.2: GDS High Level Infrastructure

Figure 3.2 shows some concepts of the system design and demonstrates functionality that is covered by the system. The GDS system design can be captured as a set of interactive layers as presented on the Figure 3.2. The main idea of this figure is to highlight multilayer organization of the system where each of these layers serve its own purpose, and which are separated between each other. The lowest two level establishes communication between nodes in the system. Nodes are both clients and data stores. Each node, when joining the system, declares its status and add itself to corresponding subscription group. There are several subscription abstraction, among them client, sequencer.

To maintain the total ordering a special subscription group is reserved: sequencer group. Over the messaging middleware a distributed component is placed. It supports the data replication which guarantees the scalability and availability by means of traffic reduction over the components. On top of replication layer a data store operation layer is placed which (a) support a wide range of operation over data, e.g., insert, update, read, range queries; (b) frame client messages with necessary information needed to access the stores, hence, resolving concurrency conflicts; (c) apply a snapshot mechanism to allow safe range query re-request.

These infrastructure makes it easy to maintain and control the system. Relying
on the INET messaging provide a great advantage to prevent all possible inconsistencies and conflicts.

3.4.1 Functionality

The basic functionality provided by the GDS composed of distributed, consistent, fault-tolerant and scalable infrastructure that serve simple requests over data. Among the requests are the following: insert, update, get, range queries. In order to make a request, the client communicates with storage part through the provided API, presented in the Section 4.2.1. Each data store processes only those messages that belong to its partition (schema); therefore, all information about the partitioning is stored on the sequencer to keep track on the number of replicas that serve the data.

With this functionality is it possible to:

- Store/Retrieve the data
- Provide consistency, availability, scalability and high performance
- Leverage the high-performance message bus and in-memory datastore
- Eliminate a need for highly scalable storage hardware

To support functionality, data models, presented in the next Section 3.4.2 was chosen to serve the maximum overlap of the system requirements.

3.4.2 Data and Query Model

GDS presents a column oriented data store at the first place with the further extension to any data base provider. This made simple, as adding new database schemas and tables into the system are relatively easy and can be plugged by the API for the data store. Schemas are not flexible: new attributes can not be added at any time but only at creating the table, as the data is stored in a fixed size column fashion as described in the Section 2.3. Nevertheless, a schema update can be applied when a running LEDS server exports its historic data to a new instance in which schema is changed. During this process current messaged are recreated by rewinders.

Each data must by marked with a timestamp, to speed up further read requests and avoid inconsistencies during the updates. The timestamp for an update is serves as a version, which should be checked before making an update and this way, a timestamp consistency is guaranteed.

The query language of GDS supports selection from a single table. Updates must specify the primary key, similar to [21]. Single table queries provide very flexible access during range requests compared to distributed hash or ordered data stores, while still being restrictive compared to relational systems.
3.4.3 Read Query Support

Adaptation of the NoSQL data stores to the relational ones keeps the need for range queries. This functionality is sufficient to further maintain data processing and analysis in offline mode. In the trading environment, support for the time range querying is very important, as further, transactional and analytic processing of data are required. Main use cases are logging, extracting order history, price history, index calculation etc. All these usages dictate the necessity for the range query support. Moreover, it can be a backbone for an stable way of analyzing the data "on the fly".

There are an extensive set of works on exploring and evaluating range queries. Among the most common solutions to support range querying is special hash function usage, that preserve locality [48], different distributed index structures, like trees [13].

GDS relies on the data locality and timestamp index which is added either by the user or data store automatically. LEDS data store, described in the Section 2.3, assures that each record timestamped and therefore, look up can be improved by specifying approximate time range. Data in the store is divided into chunks, each around 100 000 records. Each chunk is indexed according to the timestamp. Records in the chunk is time indexed. This level of separation significantly reduces information lookup time.

It was decided to apply some limitation on the range query response size. Main reason for that is availability of the system, which could degrade under transmission of unlimited size range responses. The limit is set to maximum $L = 100000$ records, which is around 50MB. Before transmission of the response, quantity of records is reported to the client. If the response exceeds L, only the L first records is transmitted to the client. If it is necessary, a new additional request can be issued to retrieve missing records.

To guarantee consistency in case of additional request a simple snapshot mechanism is triggered and snipped in the Listing 3.4. As the query is sequenced together with other operations, and no updates are supported in the time oriented schema, the query response over a historic data is always consistent unless it is requested the current state information. In this case, a snapshot message, as described below, should be issued by the data store and this way it indicate the current state of the system.

The same procedure is done to guarantee consistency during the failure of TCP connection that transmit the response. Snapshot mechanism works as follows on the snippet 3.4

<table>
<thead>
<tr>
<th>Listing 3.3: Snapshot Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>1     on_range_request:</td>
</tr>
<tr>
<td>2          send(type = SNAPSHOT, empty message) // Append SNAPSHOT</td>
</tr>
<tr>
<td>3          retrieve(query) // Read the data from the store</td>
</tr>
<tr>
<td>4          send(response directly to client)</td>
</tr>
</tbody>
</table>
Snapshot mechanism is only used for the Use Case 1, described in the section 3.3.1. Approach from the snippet 3.4 guarantees that range query response will be equal whenever it is requested. This implies only due to the absence of update operation on the time oriented data schema.

### 3.4.4 Towards Consistency, Availability and Speed

The design of a system that needs to operate in a production and within strong SLA requirements of NOMX is complex. The system needs to have scalable and robust solution for failure recovery, replica synchronization concurrency and request routing. The servers must be resilient to many kinds of faults ranging from the failure of individual disks, machines or routers. GDS uses active replication, based on the produced by sequencer totally ordered stream of messages, to achieve high availability and a consistent view of the data. Shortly, it produces fully serializable ACID semantic over the data store.

To do so, the following is used:

- for consistency, reliable totally ordered stream of messages produced by sequencer is used;
- for availability, a highly robust and fast NOMX message bus is used to support a great number of incoming operations and active replication is implemented to reduce the load from the single replica;
- for speed, a highly robust and fast NOMX message bus is used.

It is not hard to notice that all, consistency, availability and performance, depend on NOMX message middleware. This subsystem, which various functionality, leverages sustainable behavior of the GDS system, is very critical.

To guarantee consistency during update operation over object oriented schema a timestamp concurrency control is used. The following snippet shows a pseudocode for the a concurrent update resolution:

```plaintext
Listing 3.4: Concurrent Updates

on_update:
    receive(key, attribute, value, version)
    retrieve current version of a requested object
    if (obj.version == version)
        update(key, attribute, value)
    else
        reject
```
3.4.5 Low Latency

Latency is a critical part of the production oriented system architecture. However, making latency a first order constraint in the architecture is not very common. As resulting systems are usually heavily influenced by the failure resilience, availability, consistency problems etc.

The main question here is how to design a system that is oriented towards latency. A few reductions for the system requirements on the aggressive production environment are done:

- GDS applications does not require wide range deployment
- Localized disasters are not taken into account, however it could be adjusted be adding site replication

Here are the following steps on the way to the speed:

- Lightweight Sequencer. The sequencer in the system has a limited functionality and his main functions reduced to assigning a sequence number to messages and forwarding them to all subscribers. Moreover, sequencer completely isolated from the incoming message content; however, it can add additional information to the message, such as, sequenced number, other user information.

- Good Decomposition. Decomposition of the application is very important during the design of any distributed application. GDS exposes relatively decent decoupling in the system with several levels and components. The roles in the system are sequencer, clients, data stores. All of them replicated and easily replaceable. Moreover, a layer of abstraction is placed under both clients and data stores, which manages registration, communication with sequencer and makes it transparent for both clients and stores.

- Asynchronous Interactions. All interaction in the system is based on a well-known event-driven paradigm and rely on the asynchronous communication using UDP. The underlying messaging system, that uses MoldUDP, made the communication reliable. Moreover, if the necessity to rely on synchronous API appears, it is very easy to maintain it from the asynchronous API.

- Non Monolithic Data. The whole system is supposed to be stored in the column oriented storage and partitioned both by range and hash for different data sets, respectively. This gives the effect of highly decomposed data without any need to perform join, which are not supported by the system.

- Low Latency Reliable Totally Ordered Message Bus. To improve the performance a highly scalable and fast NOMX messaging middleware was leveraged in many ways. More details on it in the previous Section 3.4.4.
3.4. SYSTEM DESIGN

- *Effective programming techniques.* Following the advises from the [17, 39], GDS was build to reduce all possible overheads from the initialization, communication, garbage collection.

3.4.6 Partitioning

To scale our replication scheme and maximize performance of the datastore, data partitioning is applied both as range and hash. Use cases, described in Section 3.3, define partitioning that is specific to the column-based systems. In such system data placement is leaded by the column grouping and all record’s fields are stored column oriented fashion on disk, as shown in the Section 2.3. Column-based storage supports efficient access to an entire record and provide fast range retrieval. The former is reached because each table in the data store has defined number of rows, hence allowing reading a pieces of record from defined positions in memory. The later is guaranteed by the column orientation of the storing. This implies low latency range lookups, fast appends and well suited for a few records access. Additionally, time range reads are well served by such implementation. All these implications cover the usage scenarios and feasible to evaluate the prototype against other distributed data stores.

It is recommended to partition the data to obtain better performance. Therefore two types of partitioning are applied to the two schemes types, respectively. Time oriented schemes are object and time partitioned, while object oriented schemes are partitioned by the range and hash, which is called composite partitioning. Range partitioning maps data to partitions based on time stamp and table purpose. For example, a data for a GOOG stock during the current year is stored to one particular table.

Hash partitioning allows system to equally distribute the load and data evenly across machiness. Additional range partitioning can be applied to optimize record search. For example, every message issued during this month is stored based on their hash over four machined, as shown on the Figure 3.3(b).

All information about the nodes and their serving tables are stored and replicated on the sequencer and its stand by. This information can be used as a recommendation for the future added nodes, e.g., schemas of the failed nodes can be assigned to the new nodes.

3.4.7 Replication

Data is generally replicated to enhance two problems: reliability and performance. We are targeting both of them. Simultaneously, the major problem of replication, e.g., inconsistency between replicas, is overcome by assurance that all replicas deliver the same messages in the total order. As group communication primitive, described in Section 2.1, is the foundation of the GDS and has high performance, it was decided to rely replication on it. The basic idea behind it is to use reliable total order multicast which makes transparent all conflict operation over replicas. In
comparison to other active replication techniques, e.g., eager primary copy or update everywhere [53], data replication based on the atomic total order multicast lacks expensive coordination phase, as it is shown on the Figure 3.4. Client messages are forwarded to all receivers through sequencer. In the this project, passive replication was not considered as an option as it does not provide strong consistency guarantees without extra adjustments.

From the point of performance, replication plays a great role. As data store can both, do insert and retrieve data ranges simultaneously, certain level of replication reduces the load on one data node. This way, while first node is processing the range query, second might store incoming requests or respond to a query request from another user. Before the first node is available for further queries, it will process all relevant messages, that arrived during its range query transfer, and this will put the first node to the same state with the other replicas.

Following is the summary for the replication section:

- Replication: asynchronous, reliable, active;
- Scalability: data is organized in the range and hash partitioned tables, each with its replicated log stored on the replicas and backed up by rewinders.
- Availability: replication is implemented to redistribute the load on the request processing by the nodes; Moreover, a forward process can be deployed to broadcast messages between sites to prevent site failure, as it is shown on the Figure 3.4.

### 3.4.8 Failure Scenarios

Distributed system can be described through three main primitives: processes, links between them and messages transmitted between processes through links, as it is
In this section the implication of specific system part failures and how they are handled is discussed. At the beginning of the section some non handled failure were highlighted. Moreover, the GDS has not been designed to handle Byzantine behavior although this can be a part of the future research. The distributed data store is presumed to operate in a secure environment, where corruption of data or nodes attacks is nearly impossible.

**Client Failure**

Independently from the client failures, if the messages are issued and reached the sequencer, they will be propagated to the data store. Possible conflict resolutions are shown on the Figure 3.6.

In the first case, the message was sent and it was either lost on the way or dropped by sequencer. In both of these cases message is not delivered to the sequencer and therefore, no one delivered that message and no acknowledgment is received by the client. Note, acknowledgment here is a sent client’s message that was multicasted by a sequencer. This way, it is obvious that message was delivered.
by a sequencer and, therefore, multicasted to others.

In the second case, a message was delivered to the sequencer and then multicast to all nodes, however, a client failure prevent it from acknowledge delivery. After some time, when a new broadcast is issued by a sequencer (with a higher sequence number respectively), the client expect to get a message with the next sequence number \( n \), but it receives a message with the next number \( n+1 \). In such a case, a rewinder is requested to fill the message gap.

**Sequencer Failure**

Reliable and robust system presumes a certain level of redundancy on each level. A standby version of the sequencer is deployed in GSD. All the messages issued by the primary sequencer are received by its standby. Possible conflict resolutions are shown on the Figure 3.7.

In the first case a message \( m \) is sent before sequencer failure. Standby sequencer is informed of primary failure by heartbeat timeout. Consequently, standby reacts immediately and registers to the primary multicast group, in other words, it be-
3.4. SYSTEM DESIGN

It comes primary. \( m \) is lost, e.g., no one in the system has seen the message, and no acknowledgment is issued to the client. This does not violate consistency and a message can be resubmitted by a client.

In the second case, a message is sent by a client, delivered by a sequencer and multicasted to all nodes. However, it is not delivered by a client due to the link failure. Resolution of this situation is similar to the second case in Figure 3.6, where lost message is rewound upon gap discovery. Link failure for only some of the nodes are highly unlikely as all nodes are placed on one switch and the failure to deliver a message for one node means the failure to deliver the message for all of them and vice versa.

In the third case, similarly to the previous case, a standby declare itself a primary, however, the last message is partially lost and not delivered by a standby. As a synchronization stage takes place during primary "nomination", a lost message is recovered from a rewinder.

![Figure 3.7: Possible sequencer failure and its resolution](image)

**Data Store Client Fails**

A single data store client may crash independently and isolated from all other data store nodes. This failure does not impact any other data store client or component in the system, except that alive replicas will serve extra load until crash is overcome. The sequencer keeps track of the stores and will assign partition of the failed node to the new arrived server. Moreover, the data store is re-initialized with the all the states circled in the system previously, so all pending requests will be processed by this node together with previously processes ones. This property is served by the rewinders installed in the system that keep messages history.

**TCP Connection Failure**

During the range response transmission the established TCP connection can failure and only part of the requested records are delivered. An optimization is made to reduce the amount of data transfer on request and minimize the loses during the crash. A special message, called snapshot, is issued before the data is started to be transferred to the user, as explained in the Section 3.4. This message indicates the consistent view to the data at the moment of range query transmission. More specifically, it fixes the last seen insert message. That is why, in case of
TCP connection failure, re-request of not transmitted records can be made with a following consistent response from the server, starting from the first not transmitted row till the last record included to the snapshot.

3.5 Summary

GDS ia a unique distributed system build on top of the reliable total order multicast messaging middleware developed in-house by NOMX. It is build to serve a large amount of requests per second and perform it fast, with consistency, fault-tolerance and scalability in mind. Moreover, it is supplemented with a performance of the NOMX messaging system.

A wide set of operation is supported over the data, such as insert, read, range query, update. Moreover this set is spread over two different data sets: immutable log and mutable object records, which are actively replication by the total order stream of messages from the sequencer. Over the immutable data two types of operation are supported: insert and range query. Mutable data supports three operations: insert, update and get. First subset is made reliable by the additional fault-resilient, e.g., link failure during range response transmission, and this case of failure is very important for the exchange environment. Second subset provides resolution for the concurrent updates, e.g., timestamp consistency. Depending of the data type, the data is partitioned either by range or hash, respectively, to guarantee the maximum performance of the subset operation.

Further chapters describe the architecture of the system and show the proof of concept for performance, scalability and failure resilience properties of the prototype system.
Chapter 4
Software Design

4.1 Architecture

Entirety of the GDS software architecture is captured by a template proposed in [35]. This work describes how to present an architecture of software systems based on the multiple views, supported by optional usage scenarios. In this section the following view are presented: logical view, demonstrating static organization of the environment; process view, illustrating concurrency and synchronization aspects of the design; physical view, illustrating distributed software-hardware mapping; development view (in terms of class diagram) is decided to be avoided as it is varies a lot over time and does not reveal the necessary understanding of how the system works. Instead, an activity diagram is presented to specify implementation details.

4.1.1 Logical View

To acquire the global logical view to the GDS storage system Figure 4.1 shows the stages of request propagation. Data from the information sources is aggregated in a user application and stored for future propagation. Request messages in GDS system first has to be framed with additional information which is transparent to the user application. It is done to reduce complexity of the user applications. Additional information includes: internal message identifier, user identifier (they together form a unique identification of the message in the system), request type, optional ip/port information depending on the request type etc. After being packed, the message is sent to sequencer where it is sequenced with a unique number. As a sequencer is one threaded, all incoming messages arrive sequentially and ordered with sequencer incremental time. Messages are then multicasted to all nodes in the system and accepted only by appropriate data stores, where the message is parsed. Finally, depending on the request type, the message data is either stored or retrieved from the store. In case of data retrieval, it is further forwarded to the client directly as a stream of records.

When the cycle is completed for a data retrieval a set of further requests can be issued. Consequently, retrieving operations are blocking in GDS and therefore, it
Figure 4.1: Logical view of the GDS architecture

is required to either receive the data or timeout to continue issuing requests. This problem should be kept in mind while constructing a client application. For example, two separate processes for inserting and retrieving data can be used. Moreover, an asynchronous nature for the read operations can be easily adjusted and it was avoided only for the prototype purpose and testing simplicity.

4.1.2 Process View

GDS is build in asynchronous fashion and relies on a well-known event driven programming paradigm. Processes here are composed of two threads which are responsible for sending and listening for incoming messages respectively. As it is presented on the Figure 4.2 when one thread sends messages to the system another receives acknowledgments for its requests. A user does not communicate with the data store directly, it sends its request through a GDS_Client which has access to the message stream and sequencer. GDS_Client also follows event driven paradigm and dedicates the second thread for listening of incoming events. After being delivered to the sequencer where an ordering phase from the logical view 4.1 is happening, a message is multicasted to all participants. Arrival of the incoming data to the storage side is managed by a so called GDS_DS_Client. It parses the message, filters the request and extracts relevant records for further data storing or retriev-
4.1. ARCHITECTURE

If a read/query request is issued, the response to it is forwarded directly from the GDS_DS_Client to the GDS_Client, avoiding the sequencer.

Although the retrieval operations in GDS were made blocking on the application side, this is not strictly necessary due to its event-driven orientation. Nevertheless, it is assumed that all read requests are blocking for testing simplicity. Blocking means that no new read requests can be issued until the current one is served or timed out, however, incoming acknowledgments can be accepted.

![Diagram of the GDS architecture](image)

Figure 4.2: Process View of the GDS architecture

4.1.3 Physical View

GDS system depends on three main components: a GDS_Client, sequencer, a GDS_DS_Client. None of these are required to run on the same machine, moreover, it is actually required to run on different machines to guarantee better performance. It is assumed that they are located within a secured network. The only component that potentially faces external environments is the GDS_Client with which client applications indirectly communicate to send requests. Currently, client applications and the GDS_Client, GDS_DS_Client and Leds, the sequencer and rewinder are placed on the same machine respectively.

Both data store and client components are places on the message bus and on one switch, as it is shown on the Figure 4.3. For our experimental purposes a standby sequencer was not included in the prototype (due to the limited number of testing nodes), but it can be simply added, this way providing stronger availability and failure resilience guarantees. Communication between the client and the data store for data retrieval relies over Soup TCP, described in the Section 2.1.2.
4.1.4 Activity Diagrams

According to the process view four main components of the system is presented on the Figures 4.4 and 4.5. It can be noticed from the graphs that all processes are infinite. As it is shown on the Figure 4.4(a) a user process performs two basic operations: issue requests to the store and receive responses via callbacks. Both of these actions happen concurrently.

Sequencer is presented as a lightweight process which only received messages, assign them a sequenced number and multicast the message to all participants as it is shown on the Figure 4.4(b). Stand by sequencer behave just as any simple client of the message stream. It listens to the multicast messages from the primary sequencer and stores them.

Data stores are the receivers of the multicast messages from the sequencer and, depending on the type of request, perform different action, e.g. appending records to the store or forwarding retrieved records to the user through GDS_client respectively. To prevent any inconsistency, a snapshot message is issued. This message is forwarded through the sequencer to data store back and appended to the table. This way, any further inserts are prevented to be included in the response. In other words, if during the time between the connection to the user and records retrieval from the store, new records are inserted, they will not be included in the response. However, this step might be considered as redundant, if requests are addressed historical data and no updates are allowed on the data (which is the case for time oriented schema and for range queries), and be avoided.

Data store communicate with user by GDS_Client, just as user communicates
4.1. ARCHITECTURE

(a) User activity diagram

(b) Sequencer activity diagram

(c) Data store activity diagram

Figure 4.4: User application, sequencer and data store activity diagrams

with the sequencer and the store. Activity diagram of the GDS_Client is presented
This process is the most complicated and does the most of the work. It start from the registration on the message stream and further continues with the processing incoming messages from user, sequencer and data store. User messages are parsed, framed with additional information and forwarded to the sequencer. Depending on the message type received from the sequencer, GDS client either acknowledge an insert optimistically to the user or retrieve a unique sequence number for the message and store it for the future processing. Message from the sequencer is identical to the message sent by the client to the sequencer (multicast message), however, it is framed with the sequence number. Comparing the stored sequence number and received number from the store, the client deliver data records from the store. Further, these records are forwarded to the user.

![GDS_Client activity diagram](image)

This infrastructure is built in mind with consistency, reliability and speed. To obtain these properties, a lightweight sequencer together with the actively replicated infrastructure gives a required result.
4.2 Implementation Details

4.2.1 Client API

The client API is quite simple and straightforward. It is focused on serving applications that require to store a large amount of data with a further search over entries. The main entry points of the API are:

- Messages are sent. A user/application invokes a send operation (append, query, insert, update, get). It is forwarded through the system and delivered to the data store, where the request is filtered and processed. Filtering is used as all messages in the system are delivered to every node and therefore, they should be addressed only by corresponding data store.

- Messages are received (as a callback). A user/application receives messages from the system, as a response to its requests and a stream of messages is received as the response to its time range query. All messages are acknowledged either optimistically, e.g. inserts, or with a response from the data storage, e.g. updates.

The client C API provides an application a transparent communication to the data store. Each message is required to have message type, sender id, message id, message content. Having both sender and message id provide global identification of the message in the system with several clients.

A following set of operation is supported:

- **Time Oriented Schema.**
  - `int gds_insert(userHandle handle, int schema, appendStruct const * message, int size);`
  - `int gds_query(userHandle handle, int schema, queryStruct const * message, int size);`

- **Object Oriented Schema.**
  - `int gds_insert(userHandle handle, int schema, insertStruct const * message, int size);`
  - `int gds_update(userHandle handle, int schema, updateStruct const * message, int size);`
  - `int gds_get(userHandle handle, int schema, getStruct const * message, int size);`

, where the `userHandle handle` is required to detect whether a user is served by the GDS_Client instance.

- **Callback-receive function.**
  `void (*gds_on_event_t)(int type, void const * message, int size);`
Following API is implemented as shown on the figures below for insert, update and query operations respectively. In GDS system, append and inserts are not considered as conflicting and that is why, delivery of the request to the sequencer is sufficient enough to guarantee an operation to occur, as shown on the Figure 4.6. On the first step the message is forwarded to the sequencer. On the second one, sequencer multicast the message to the data stores. Among receivers of the multicast message, clients and data store are both also listed. This way, if the retransmitted by sequencer message is delivered by a requester, it means that a sequencer has forwarded the request to all.

![Figure 4.6: Append and insert](image)

In the Figure 4.7 similar request is shown, except that during the update operation an acknowledgment of the operation from data store is required. In this case, sequencer is not only propagate a request message, but also it should multicast a status of the update operation, received from the data store. Pale lines indicate that sent message is ignored on a receiver side.

Finally, requests, that require data retrieval from the data store, are not transmitted as in update example. e.g., through the sequencer, but rather forwarded directly to the clients through opened TCP connection on the client side, as illustrated on the Figure 4.8. Following design was motivated by the performance optimization and reduction of the required messages to start the query transmission, that is, only 1 message is required before a requested data starts flooding the user.

![Figure 4.7: Update operation with acknowledgment](image)
4.2. IMPLEMENTATION DETAILS

4.2.2 Bootstrapping the system

When a sequencer node starts it registers as a primary sequencer in the system. Further, when other nodes/clients appear they subscribe to the sequencer publishing group and register on the sequencer as subscriber. Simple register operation is used for initialization phase in the system. At the same time, recovery nodes are started to protect the system from message loses, e.g., rewinders and standby sequencer.

Each node on the message bus is tracked by a sequencer with heartbeating. In case of recovery, all messages that were transmitted through the system are replayed from rewinders and therefore, recovered replica stays up to date with other participants.
Data stores appear in the system by starting them manually. Each data store claim its static schemas upon start and these schemas cannot be changed at a run time. If a new scheme for the application is required, a new data store with corresponding schema should be created.
Chapter 5

Evaluation

5.1 Prototype

The concept of sequencer based total ordered architecture was evaluated by developing a functional prototype. Following prototype implements three types of operation: insert, get, range query and does not include support for an object oriented schema which is a matter of future work. The prototype is built on both C++ and Java with messaging relied on INET message bus infrastructure. The whole system is based on a well known concept of event driven programming. All received messages triggers receivers `on_receive` callback function on the specifically dedicated thread, that listens to incoming messages. The GDS system is deployed and evaluated on 3 machines in a NOMX data center with 10 Gbps bandwidth connection.

The client has a C API. Data store uses Java APIs and is supplied by a configuration file which enable various customization, e.g. number of instances, replication factor, ip addresses and ports information for the subscription groups etc. In addition to the prototype, several tester applications were developed. They are developed in C++ and mainly generate workloads, parse the log files and measure performance.

<table>
<thead>
<tr>
<th>Language</th>
<th>Num of Files</th>
<th>Num of Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>8</td>
<td>1793</td>
</tr>
<tr>
<td>Java</td>
<td>7</td>
<td>560</td>
</tr>
</tbody>
</table>

The following evaluation shows that the prototype is able to serve the required number of requests per second and satisfy the latency criteria. Further improvement are discussed in the Section 6.3.

Moreover, the data store has proved to be scalable and fault-tolerant due to the reliable uniform total order multicast primitive developed by NOMX. At the same time consistency of the data store can be proven as follows below.

**Consistency of the GDS 1.** The GDS system based on the reliable uniform total order multicast is consistent for all types of operations over the data under failures.
Proof. By definition \(^1\), consistency is a property that ensures that any changes to a value in an instance of a database are consistent with changes to other values in the same instance.

As all the messages arrive at the same order to all the nodes, violation of the consistency property within a properly functioning system is impossible. The only possibility for inconsistency to take place is concurrent updates and possible failure scenarios. As described in Section 3.4.4, concurrent updates are managed in the system by timestamp concurrency control where each update request contains the last seen version/timestamp of an object.

To avoid any possible inconsistencies during failures, as described in the Section 3.4.8, a certain level of replication is introduced to the system. Active replication and properly configured rewinders prevent GDS from losses of critical information.

The evaluation is composed of three types of tests: performance, scalability, fault-tolerance. While performance is critical due to the large number of requests to the system, it is equally important to demonstrate how the system scales and performs under failures.

For scalability and performance measures of the throughput and latency for different numbers of instances were measured. These tests are performed over synthetically generated data.

Evaluation of the prototype was made over 3 machines in the NOMX local data center. Figure 5.1 shows the distribution of the running processes over specific machines. All clients and Leds servers were deployed on one 12 cores machine where a sequencer and rewinder were placed into the machine with 4 cores. All machines are deployed with 10Gbs network card.

5.2 Performance

Performance of the GDS is primarily determined by two things: the sequencer and the data store requests processing. Even though the sequencer plays an intermediate role and just forwards messages to all others, every request have to pass through this component. Hence, it risks becoming a bottleneck. However in the GDS system it is replicated in multiple ways, e.g., hardware and software and that is why, it is a reliable component in the system. Moreover, it is able to serve more than a few millions requests per second.

Request processing is challenging due to the high load for both inserts and range queries. Therefore, a certain level of replication should be introduced to split the load between range response transmission and simple inserts correspondingly.

Performance was measured in five stages. First, insertion operation were challenging. Both throughput and latency are measured to evaluate system behavior

\(^1\) http://en.wikipedia.org/wiki/Consistency_(database_systems)
5.2. PERFORMANCE

Figure 5.1: Prototype Set Up

under constantly increased load and number of concurrent users. Second, latencies of single record retrieve operation are analyzed for both cases, with and without simultaneous inserts. Third, range request completion time for different concurrency level were measured. Fourth, scalability characteristics for all, insert, single record read, range read, were performed and analyzed. Fifth, failure scenarios performance for range reads were tested. All tests were made for maximum of 6 concurrent users data stores (schemas). Data store read operation were measured over a sub-10 million record data store.

System time in nanoseconds is used in all the tests for the latency measurements. Throughput is simply measured by incremental counting of the incoming messages. Performance measures are made for average throughput and latency indicators. A load generating script was developed to manipulate both inserts and reads.

5.2.1 Inserts

To determine the maximum throughput of the GDS within defined set up, throughputs test was designed. During this test number of concurrent users was increased from 1 to 6 where each user’s request rate is around 75 000 requests per second while number of data stores remains equal, e.g. 6. Each message was sized to 120 bytes. Number of concurrent users is limited to the six due to the limitation of physical testing environment where both users and GDS_Clients were deployed on one 12 cores machine as different processes.

As it is shown on the Figure 5.2, the number of concurrent users in the system
increased from 1 to 6. Number of leds nodes is increasing correspondingly to number of concurrent users. As seen the throughput grows almost linearly until it reaches over 400 000 inserts per second (appends). That is the average throughput is proportional to the number of users.

Figure 5.2: Insert operations throughput for 100 bytes messages increasing number of concurrent users

To explore the limits of the data store, more intensive load was generated with double and triple concurrent users simultaneously per each data store respectively. As can be found on the Figure 5.3, increasing the number of concurrent users stabilize GDS throughput which also is the maximum serving capacity for Leds nodes. A small reduction of scalability for 4 nodes with 8 and 12 concurrent users on the figure correspondingly is due to the load generators placement, that is, on the same node. All tests ran until each concurrent user stores 10 000 000 requests to the data store and it took on average 2 minutes to complete.

As it is very important for exchanges to have a fast acknowledgment of the insert operation latencies were measured and are presented on the Figure 5.4. This figure plots minimum, average and maximum latency for different load. In this test the number of concurrent users were increased from 1 to 6 with 75 000 requests per second each. Latency for a single record acknowledgment varies from 50 to 300 microseconds for load from 75 000 to 450 000 requests per second respectively. The average index increases from sub-100 microseconds latency for 2 concurrent users to sub-150 microseconds for 5,6 users.

As expected when running on the limited set of machines the performance is bounded to the maximum capacity of the stores and machines. The acknowledgment, albeit optimistic, guarantees delivery of the message to the recipient in the
5.2. PERFORMANCE

![Figure 5.3](image)

**Figure 5.3:** Data nodes scalability for insert operation with twice and trice number of requests per node

![Figure 5.4](image)

**Figure 5.4:** Insert operations minimum, maximum and average latency depending on the number of concurrent users in the system

same order with other nodes. Therefore, it is sufficient enough to ensure the client in data durability.

To sum up, the data store is able to scale linearly to serve inserts over 6 Leds servers for 1 to 6 concurrent users each 75 000 requests per second. Inserts produce
a sub-150 latency and are stored durably to the stores.

### 5.2.2 Get - Single Record Read

Even though single record read is not intended to be served by the time oriented schema it was still interesting to discover the latencies for the single record retrieval. Two sets of tests were made for the get operation, e.g., get with and without additional insertion load over a sub-10 millions record database. In the first case, as shown on the Figure 5.5, a set of read requests were made from 1-5 users respectively. Each client issues 100 random reads to 5 Leds nodes one after another. Each response record is 120 bytes. Figure 5.5 shows a double increase in latency from 1 to 6 users from 250 to 500 microseconds. The main reason for this behavior is that the number of records to lookup increases proportionally to the number of concurrent users.

The second test was performed with an additional insertion load of 10 000 messages per second and divided across the data stores at random. As presented on the Figure 5.6, the latency is slightly increased in comparison to the first test without extra load, however it is still within 500 microsecond response.

Unlike insert operations, the latency here composed from the time between the request is sent and a response is delivered from the data store. This interval includes the following: time from the client to sequencer, time for a sequencer to multicast a message to all participants, time for filtration on the store side, time to retrieve the record and time to send the response to the client. Further optimization of each particular component of the latency is desirable to reach the maximum performance. The main purpose of this tests is to show that the get performance characteristics

![Figure 5.5: Get operation latency over filled data store](image-url)
5.2. PERFORMANCE

Figure 5.6: Get operation latency with an additional load of 10,000 inserts per second

are satisfactory enough even on the prototype version of time oriented schema which was not optimized for single record lookups.

As for the purpose of the prototype the single record reads were made blocking, the scalability of the system is presented in terms of latency over different number of Leds instances and is shown on the Figure 5.7. In this test the number of Leds instances varies from 1 to 4 and the number of concurrent users is fixed to 3 with 100 get requests each. In other words, three concurrent users request random records from a random Leds servers and the number of Leds nodes varies from 1 to 4. Experiment shows that increasing the number of data nodes improves latency for a get operation form 400 to 250 microseconds over one to four data nodes respectively. Latency improvement is around 40% for 4 data nodes.

5.2.3 Range Query

Figure 5.8 shows the results of the range query requests performance test. In this test number of both users and data store varied from 1 to 4. Independently on the number of data store each user issued 25 query request with 100,000 records response from a 10 million records data base. The main conclusion that can be made from this graph is that the system scales on adding extra users and hence, increasing the number of records that can be retrieved from the store. The trend line shows a good linear scalability already for 4 data stores and clients. Moreover, the 350,000 records per second throughput is the maximum number of records that can be delivered by 4 concurrent users (as each request made blocking by the user side).
Figure 5.7: Scalability of a get operation in terms of latency reduction over multiple data nodes.

Figure 5.8: Range query throughput and scalability.

However, the stability of throughput for 1 concurrent user while increasing the quantity of the stores influenced by the blocking nature of the query request, e.g. no new queries are sent until previous one is served or timed out. On the other hand, increasing number of users shows us the maximum possible capacity of 1 Leds instance which is approximately 130 000 records per second.
According to the Section 3.2 the data store capacity of the response stream should be 300 Mbps. This test showed full satisfaction of this requirement. The calculation are the following for the message size of 120 bytes:

\[
300\text{Mbps} = 37, 5\text{MBps} = 312500
\]

requests of size 120 bytes per second. Even though that the performance results are satisfactory, further optimization for the query retrieval should take place, i.e. the possibility to specify time range for a specific data range improves look up time.

![Figure 5.9: Range query scalability under constant load of 8 concurrent users for different replication factor](image)

The most interesting results are presented on the Figure 5.10. This graphs shows the scalability property for the concurrent range query requests from 8 simultaneous users. The system shows linear scalability while adding replicas. Throughput starts with the 140 000 records (each 120 bytes) per second for one machine and corresponds to the near maximum capacity of one store. Additional replica introduce doubling the throughput to 340 000 records per second. Finally, 4 replicas can look up and deliver near 600 000 records per second for 8 concurrent users.

Such a speed up is caused by the natural load balancing in the system where each replica can serve different clients. As it is shown on the Figure 4.8 while first replica transfer records to one clients, second replica may serve another client.
Failure of the data store is transparent to the user due to the use of active replication. Failure of the sequencer limits system availability until the new sequencer reappear which takes less that a few seconds. However, the failure of the query response transfer (TCP connection failure, data store failure) is noticeable for the user and require re-request of the query.

To optimize the consequences of the possible range response transfer, when a failure detected, a new request can be issued to retrieve only missing records by specifying the first_record to return. By default the first_record is set to 1 as it is required to retrieve the whole set, nevertheless, it can be modified by the user on failure.

This performance optimization gives the following improvement as shown on the Figure 5.10. The first column shows the total time for the 25 range queries with 100,000 records each(each message size is 120 bytes). The second column represent the completion time for 25 queries under 5 failures (5 transfers were interrupted and have to be fully re-requested) and showed the completion time increase of 35%. Finally, the third column corresponds to our proposed optimization and gives an improvement of 15% over the simple approach (in this case only missed records were re-requested from the data store).
Chapter 6

Discussion

The report brings together a set of concepts to approach a performance characteristics of the distributed data store based on uniform reliable total order multicast. There is a bias associated to use different data sets exclusively. Therefore, two data sets are supported: log and record based. This differentiation allows to tune a performance over two sets of operations, concentrated over range queries and concurrent updates respectively. This section is organized as follows. Firstly, main findings are discussed. Further, positioning of the GDS among other large scale data stores is highlighted and main advantages are pointed out. Finally, further research is identified.

6.1 Main findings

6.1.1 Performance

From the performance results on throughput and latency it is known that the system safely handles 400 000 requests per second with 95th percentile latency at a 100 micro seconds using 6 nodes. The results surpass the performance requirements specified in the Section 3.2. However, the performance can be improved even further.

In this section some alternative setup and improvements are discussed. The findings are based on the scalability experiments presented in the section 5.2 as well as it was discovered during the prototyping.

- **Speed:** Performance evaluation showed complete satisfaction of the SLA requirements needed to serve unique stock exchange environment. Several workloads were used to explore the GDS performance, e.g., combination of inserts, reads, range reads. The system benefits from the time oriented storing in two ways: fast appends of the data, faster lookups for the time range. Moreover, relying on the NOMX INET messaging system, inserts can be acknowledged not by a data store but by the indication that sequencer received the request. This way the message acknowledgment narrow down to the client-sequencer round trip. The findings of this evaluation are restricted to the synthetically
generated data and are a matter of the future work where the system should be tested over real world data.

- **Adding Extra Nodes:** Despite not initially being considered a bottleneck, the sequencer’s multicast might show some overhead on the data store performance as all the messages are forwarded to all nodes. It means that extra messages are needed to be processed and further discarded instead of direct request processing. Consequently, adding extra nodes will increase the number of requests served, as the amount of different data stores in the system increases; however, the message pressure on the data store nodes will grow proportionally. An alternative within the infrastructure can be applied by splitting the system into separated GDS systems that serve independently different schemas.

Furthermore, all records are stored in memory and organized in tables with fixed size. Upon filling the table, it is compressed and decompression is required to retrieve those records. Moreover, when it is necessary, compressed chunks can be managed to disk in the background. Optimization for the table size is required to minimize the overhead from compressing and retrieving information from the disc. It is suggested to apply machine learning techniques to optimize table size. For that, historical data with different table size and performance overhead.

### 6.1.2 Consistency, availability, failure resilience

Reliable total order system is leveraged to maintain consistent view of the data in any given point in time. **Consistency** is assumed as all the processes in the system deliver all messages/requests in the same order. Therefore, no matter which data store responses to get request, this operation is served only after all previous writes are seen. Moreover, any inconsistency and not-in-order messages are detected immediately and re-requested from rewinders, that store all the messages. The limitation of consistency statements are clear: even though that multiple level of hardware and software replication makes it impossible to violate consistency, purely theoretical scenario with failure of all components except one could introduce some inconsistency to the system. As a result, a rollback mechanism for operations should be implemented. Notwithstanding its limitations, the system guarantees consistent updates and view to its data all the time.

**Availability** is divided into two parts: access availability, the ability to perform an action no matter the workload; failure availability, transparent recovery from failures and its affects.

- **Access Availability.** As all the messages are backed up and can be replayed any time they are requested, all insert/update operation can be optimistically acknowledged and eventually they will catch up in total order. Therefore, the availability of the GDS is restrained to number and size of concurrent range query operations. As the possible range query response size is limited to the
constant number of records, the response time is restrained to the network transmission rate. However, a minor lack of availability can be presented during the sequencer failure, where no requests are served until a stand-by replica appears as a primary.

- **Failure Availability.** Tolerate failures and recover from them is a principal challenge in designing consistent and available systems. GDS can handle both host and link failures by replication, range query re-requests and absence of host roles, e.g., primary, stand by data node. Moreover, a multiple redundancy is implied for sequencer and rewinders. All of these, joint to reliable total order baseline, guarantee a transparent resolution of failures and conflicts.

### 6.1.3 Summary

Building a distributed storage system is a heterogeneous problem. It includes designing the architecture that satisfies multiple criteria and simultaneously adjust them to guarantee maximum possible throughput and minimum latency. Additionally, it was found that (a) maintaining both consistent, reliable and highly performance system is very challenging and require innovative design solutions, (b) lightweight sequencer implementation has allowed to maximally benefit from the INET messaging infrastructure, (c) performance characteristics of the proposed solutions are satisfy the SLA requirement and can handle over hundreds of thousands requests per second with sub-100 microseconds latency.

### 6.2 Positioning

#### 6.2.1 Comparison to existing systems

Very few system uses a reliable total order multicast as a part of its underlying infrastructure [31]. Moreover this usage is bounded to the limited application, e.g., replication only. [44] compares the performance of two known approaches to implement total order and suggests a new approach, combining TO different implementation techniques. In addition, the most famous system based on TO was considered failure prone [33]. Perhaps because of these problems, joint with scalability and implementation challenges, total order reliable baseline have not made it into more large-scale distributed systems. Given the innovative NOMX software and hardware development and usage respectively, that is more suited for maintaining reliable, consistent, available infrastructure, previous challenges become less of a problem.

The prototype that was developed is intended to replace currently cost expensive solutions on the stock exchange market. The system is solving both problems of performance with being consistent, failure robust and available. Moreover, the former is optimized to serve main operations necessary for the NOMX, e.g., insert, range lookup, get.
Comparing other data storage system to the GDS, GDS tries to maximize amount of guarantees it can provide, excluding transactions and complex queries support.

Calvin layer [50] over the storage system might be the system most similar to GDS prototype in terms of underlying concepts. Both systems are intended to serve as a layer above the storage system to handle operations replication and network communication. However, GDS is made simple by using only one isolated and replicated sequencer, instead of multiple sequencer components on each node with necessary synchronization over replicas. Despite critique towards single sequencer approach, e.g. single point of failure and scalability, GDS tries to approach them. Former is targeted by replicating the sequencer while latter is addressed by the NOMX hardware solutions. Moreover, GDS is based on the communication system used and proved in production and that is why has advanced pure state of the art Calvin system. From the purpose point of view, StockDB [5] is related to the GDS by optimizing appends and range retrieves. However, it mainly focuses on the data compression optimization and lacks flexibility in range retrievals, where only compressed chunks of defined equal sizes can be extracted.

6.2.2 Stream Processing Applicability

Current data stream solutions are based on the off-line stream analysis, e.g. data is pulled out from the data storages by MapReduce and processed remotely off-line. Some of the examples is a BigTable, PNUTS, Hadoop, HBase etc.

However for some application on-line processing is required. Among existing systems are S4, Spark, Storm [40, 55, 6]. Leverage of the GDS for the on-line stream processing is possible and made simple by the underlying reliable total order multicast protocols. Simple operations, such as aggregation, can be performed on the fly and stored to the data store again.

Figure 6.1 shows an abstract example of the stream processing within GDS infrastructure. GDS can host not only the data but also the applications that act on and refine the data. For example, a clearing system where the first phase of input data consists of the trade executions and underlying prices, the second phase of data consists of the member positions, the third phase of data consists of the margin values, the fourth phase consists of the member deliveries and payments, and so on. For each phase, a subset of the active servers react to the data by their callbacks. Moreover, as all the data is persisted on the stream (by rewinders), the application callbacks have no responsibility to persist the data for durability, unless it is necessary for the application for further post processing of the data.
6.3 Future Work

6.3.1 System Improvements

- **Generality**: Generality is not plugged currently and therefore different tables with different data types should be specified explicitly. The way generality can be proposed is to introduce JSON objects as a way to exchange data, which are parsed on the data store level. Also an automation of schema propagation should be adjusted. So that all clients are notified upon appearance of a new schema/partition.

- **Compressed results**: To improve the performance of the range queries retrieval, instead of record-by-record transfer, a compressed set of records might be transferred directly to the client, reducing the amount of messages sent to the client and reducing the size of the response.

- **Operations rollback**: The system might lack the delete operation. It means that it is not possible to simply rollback if it is needed and the deletion should be handled by the application. This can be approached by issuing an update to the record with 'deleted' status.

- **Implementation adjustments**: Implementation optimization includes complete avoiding of the garbage collection while storing the data and while currently takes place. Also, an asynchronous non-blocking nature of the read requests should be used for the real environment. Currently read requests are
blocked by the user/application, so that, it is not allowed to issue new request until the current is served or timed out.

- **Partitions coordinator:** Automated balanced partitioning of the data store across all data servers is a great improvement to simplify the maintenance. The main purpose of the coordinator is to ensure balanced key/schema distribution across the servers and facilitate membership changes as servers may leave and join the system. Moreover, this component is not on a critical path as no requests is flowing through it.

- **Long running tests:** The performance tests shown in Chapter 5 are mostly short-lived. Longer-running tests are required to see how the performance is affected over time.

### 6.3.2 Research Directions

In the bigger picture and from an academic prospective, the data base research community would benefit from the extensive survey on the distributed data store properties (consistency, availability, failure resilience etc) and its fulfillment by different implementations. Moreover, a list of techniques that can be used to reach one or another property might be very helpful for those who are trying to improve current solutions. Regarding GDS, an interesting directions could be an extensive performance exploration with unified comparison (e.g. the same workloads, partitioning, set up etc.) to other existing solutions. Finally, integration with the other datastores, such as PostgreSQL, is an interesting endeavor, as the GDS may combine multiple databases in one place.
Chapter 7

Conclusions

High performance distributed data store systems are becoming an important component in the exchange systems where a huge churn of information should be logged/stored durably and reliably. Especially, it becomes important after the new auditing requirements were laid upon the exchanges, such as storing all orders and trades for many years. Real time, fast and reliable storing and retrieving of the data for further processing is a great endeavor. However, reaching the SLA requirements within a real time conditions is not straightforward. A variety of data store solutions has been proposed to address the problem of either consistency, availability or performance. While most of the previous system has focused on the complex consensus protocols to guarantee strong consistency and fault-tolerance, or solely performance, little exploration on uniform reliable total order multicast primitive to satisfy exchange environment has been provided.

In this study a design for a real-time, large-scale, low latency, reliable and consistent distributed storage system has been proposed. The backbone of a system is a uniform reliable total order multicast messaging middleware, developed internally by Nasdaq OMX, which both provide a great baseline for fault-tolerance and performance. The system composed from three main components, i.e. clients, sequencer, data nodes. To guarantee fault-tolerance, a certain level of redundancy for both sequencer and data nodes is provided. Moreover, consistency is preserved by the extensive use of rewinders, capable to rewind lost messages for the participants. All data is stored in memory.

As such, GDS system can serve near half a million write requests per second over 6 data nodes. Experimental results show that an average latency for write operations varies within sub-100 micro seconds with a maximum sustained throughput of approximately 400 000 requests per second. Read operation are shown to perform within sub-300 - 500 microseconds for a sub-million records data tables without and with addition load respectively. Range read operations are appeared to be served with 300 Mbps capacity over 4 data node instances. All writes and reads show near linear scalability by adding extra data nodes. Theses results are taken from deployment of three nodes.
In conclusion, this report shows promising results to scale the system for real time, strongly consistent and reliable data processing. Moreover, the design of the system promises to be easily extensible for real time on-line data processing and analysis.
Bibliography


